Winston Cooper

3/20/19

Prof. Pang

CMPS 161 Final Project

Title: WordSource: A Word and Sentence Structure Visualization

Abstract: This project aims to provide tools for the analysis and generation of text patterns found within a body of text.

Introduction: The human language is a critical part of modern society; it's what defines us as human and sets us apart from the rest of the life on Earth. Everyone deals with some form of language on a regular basis, but most of the time it passes us by without second thought. This project, WordSource, aims to allow users to discover patterns that occur within language, which heavily vary from speaker to speaker. By providing multiple visualizations of tokens in a provided text, we can quickly determine the subject matter, as well as see trends that a speaker tends to follow.

Related Works:

- WordTree - Concordance Visualization
  https://www.jasondavies.com/wordtree/?source=obama.inauguration.2013.txt&prefix=We
- Wordle - Word Frequency Visualization

http://www.wordle.net/

Technical Details: This project is written in Python and Javascript, utilizing Python libraries for the parsing of tokens from the text and Javascript libraries for control and the drawing of visualizations. There are five main parts to this program: The uploading and processing of user text files, token frequency bar graphs, concordance generation, WordTree generation, and probable text generation.

The technologies used in WordSource are: Web2Py, NLTK, pdfminer, chardet Vue.js, D3.js, and standard HTML/Javascript/CSS. See References for further information.

Uploading of user files is done through the Web2Py MySQL table system. The uploaded files and metadata are entered into a WordSource table for future reference on upload, and the user is brought to the Available WordSources page. When the user clicks on one of the uploaded WordSources, the specified file is retrieved and processing begins.

Text Processing is mostly done using the amazing tools provided by the Python Libraries pdfminer and NLTK. If the file is a PDF, then pdfminer is utilized to parse tokens from the PDF into one cohesive string. Otherwise, the text is in a text file and must be decoded according to its encoding. The Python library chardet is used to detect this encoding and decode the file accordingly.

Once we have a string of raw text from the WordSource, we use NLTK to tokenize the text, and convert each token to lowercase (to subvert capitalization differences). NLTK is then also used to create a frequency distribution of each token as well as a token length distribution for each token. A total number of outcomes (total tokens) is also retrieved using NLTK for

percentage processing. After the text has finished processing, the resulting array of tokens is passed back to the Javascript, specifically a Vue app which contains the information for further processing.

Once the tokens, frequency arrays, and number of outcomes are passed back, Javascript functions are used to create a WordTree and probability array. More on these later. For now, the default visualization must be drawn, the frequency bar graph. Occurrence percentages of each token is calculated and stored in a FreqDistNode object for each token, containing the token and resulting frequency. Each of these objects are then entered in an array, then sorted.

The frequency visualization displays this array of sorted FreqDistNodes by creating a simple grid and bar graph, where the domain is the token and the range is $0 \rightarrow$ max frequency of a token. The visualization is drawn using D3 and an SVG canvas. From here, a user can observe the highest frequency tokens and take away the most frequent topics of the text. The problem with this however, is that there are a lot of grammatical words in the human language (the, of, and, in, this, etc.) without much semantic value. To account for this, the user has the ability to add exclusions to any token, redrawing the frequency distribution with the new exclusion(s) and resulting max frequency. The exclusions can be reset at any time.

The second feature of the frequency visualization is the token frequency bar graph. This is very similar to the token frequency visualization except instead of the token itself, the domain is token length, in descending order. This can be used to analyse the tendency of a speaker to use longer or shorter words. In both visualizations, the user can specify the number of entries to display, fundamentally the size of the domain.

The second visualization feature of WordSource is the ability to generate concordance.

```
>>> text1.concordance("monstrous")
Displaying 11 of 11 matches:
ong the former , one was of a most monstrous size . ... This came towards us ,
ON OF THE PSALMS . " Touching that monstrous bulk of the whale or ork we have r
ll over with a heathenish array of monstrous clubs and spears . Some were thick
d as you gazed , and wondered what monstrous cannibal and savage could ever hav
that has survived the flood ; most monstrous and most mountainous ! That Himmal
they might scout at Moby Dick as a monstrous fable , or still worse and more de
th of Radney .'" CHAPTER 55 Of the monstrous Pictures of Whales . I shall ere l
ing Scenes . In connexion with the monstrous pictures of whales , I am strongly
ere to enter upon those still more monstrous stories of them which are to be fo
ght have been rummaged out of this monstrous cabinet there is no telling . But
of Whale - Bones ; for Whales of a monstrous size are oftentimes cast up dead u
>>>
```

Concordance is a list of words that are present in a text, containing the target word. This comes packaged as a utility in NLTK, which both made things easier and harder. However, the function does not return the result of the concordance. Instead, it prints it to console, so a workaround was necessary. After a quick google search, I found a function that would suffice (credits in references). A list of concordances is then passed from the Python to JavaScript via JSON response, which is drawn to the SVG using D3. While not difficult to implement by hand, the concordance was not the main focus of the visualization. There was much to be done to improve the appearance of the concordance, but I decided instead to focus on the other visualizations due to time constraints.

The third and main visualization is the WordTree visualization. This is the visualization I had in mind when I started the project. Now, we return to the WordTree and probability array that was mentioned earlier. By parsing through the entire text and creating a WordTreeNode for each unique token, we can build a tree for which each token's children are the words that occur directly after it in the text. Every time we encounter a new unique token, we create a new WordTreeNode. If a token is encountered more than once, we do not create another node, but

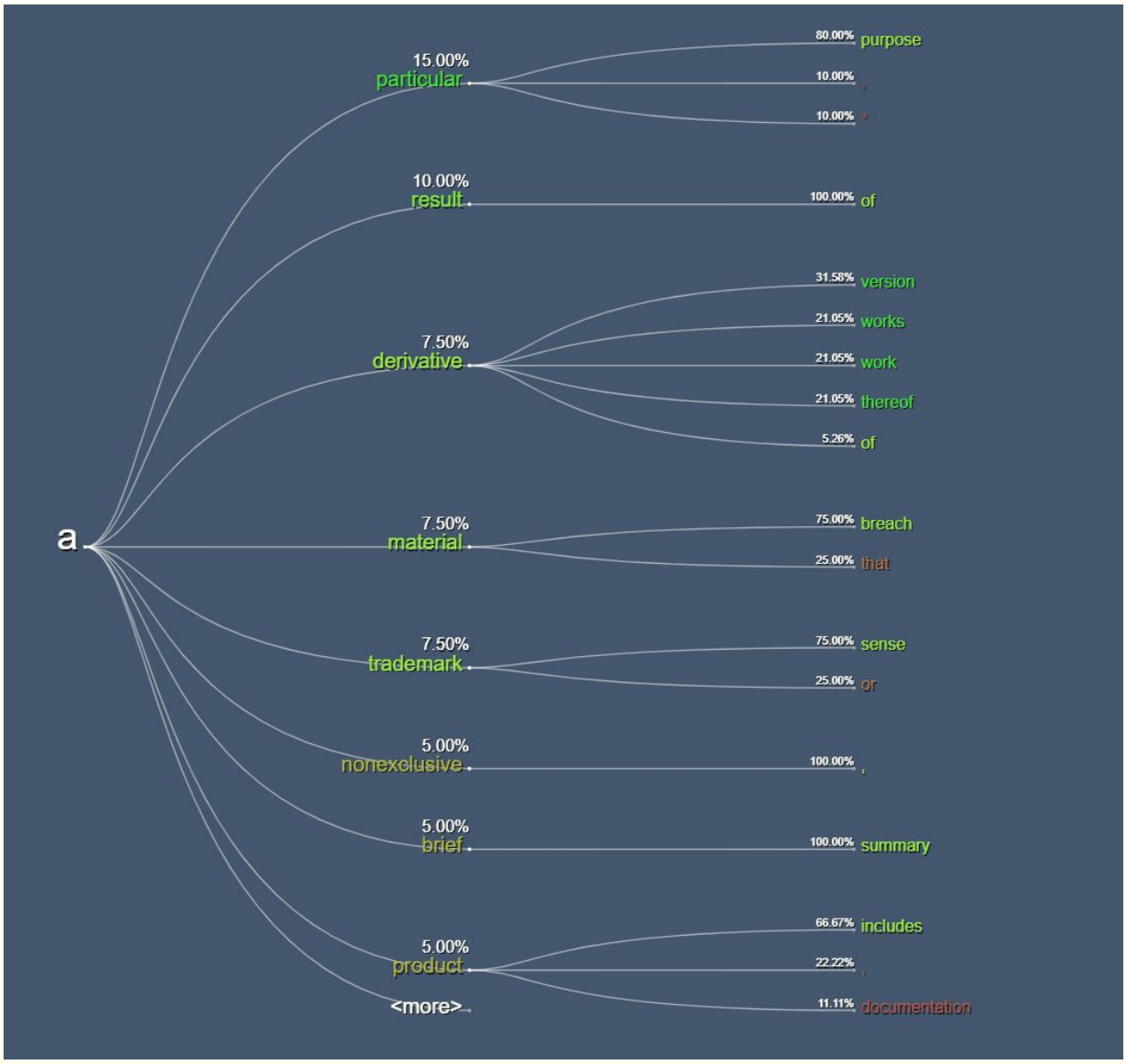
add the existing node for the token to children of the parent WordTreeNode. Once parsing of the text is finished, we have a complete tree, likely cyclical.

Now, we construct the probability array in the same parse which is to be used to determine the likelihood of a token appearing after a specific token. To do this, we create a ProbabilityNode, which has three attributes: the token that serves as the parent token, a total count of all tokens that occur after the parent (non-unique), and an array for all of these tokens. Once we have created a node, we store it in WordTreeArray, which functions almost like a hash map. To calculate the probability of a child token occurring after the parent, each unique token in the parent's token array is counted, then divided by the count to obtain the probability. The WordTreeNode and ProbabilityNode are created as separate objects because they have separate functionality. A ProbabilityNode has a function called generateNext() which can be used to randomly choose a possible next token as they occur in the text. This will later be used to generate text.

Now, for the actual visualization of the WordTree. The visualization centers around the d3.tree() function. By default, the first token encountered is set as the root. A d3.hierarchy is specified on the children, and a tree is created. There were some issues with this initially. Since there are usually cycles in the WordTree, the same object would be drawn every time it appears in the tree. This would be fine. However, setting one of those nodes to inactive would set the other instance to be inactive as well, since they are the same object. When stopping at the maximum depth of the tree, this is a problem since it would truncate the tree in the wrong place. To work around this, a new Object, DisplayNode, is created and is meant to be a copy of each WordTreeNode for display purposes.

On every displayed node on the tree, there is a click function which will set that node as the root and redraw the tree, allowing a user to traverse through the tree. Once I had the main functionality of the tree working, I implemented a second attribute of the tree, the color mapping of frequencies. By using the array of ProbabilityNodes constructed earlier, we can look up a token t and one of its children to get the percentage of its occurrence, given that we have just seen t. Then, by using a d3.scaleLinear() function, with the max frequency of any child for a token as the maximum domain, we can scale the color of each node to provide a more intuitive first look. We are fundamentally creating a huge, traversable first-order markov chain and displaying it for the user. There are additional controls for the user to specify maximum tree

depth and maximum number of children allowed, for sake of readability.



The last feature of this project is the text generation function. Since we have already constructed a markov chain using each token, the only thing left would be to generate results and display them. We return to the class function in every ProbabilityNode, generateNext() to do this. Our initial root is the same as in the WordTree: the first token to appear in the text. From there, we call generateNext() on the current token to generate the next token, and compile them

into a cohesive string, which will be displayed in the program. The user can specify a maximum token count for the generated text.

Results:

The combination of these four visualizations allows a user to gain insight into a body of text and the range of words that a speaker is likely to use. This application can be used to analyse large bodies of text, though while not completely optimal, it has been able to handle 200 page PDF's, given a short period of loading. If I had more time, I would've liked to improve the usability of the concordance generator (e.g. highlight the query, render the phrases aligned by the query token) and display it side by side the token frequency graph, for better utility. The Word Tree was the main focus of this program, and I am quite happy with how it turned out. The difference between this word tree and the word tree listed in related projects, is that this word tree seeks to display the frequency of each token's occurrence, while the other word tree focuses on concordance. There was much more that I had in mind, such as reverse traversal capability, and token search mechanics, but I sanded the program functionality down to the basics in consideration of time constraints. The results of the generated text range from eloquent to nonsensical as one would imagine. I've received a good deal of enjoyment from feeding song lyrics into the program and generating new lyrics. Further expansions on this program would see it using higher order Markov chains in the generate text to possibly create more intelligible text. Another expansion would add grammar parsing capability to the program, creating more of a grammar tree for which the generated text could operate by. As it is now, the program considers no part of semantics, only patterns. In the real world, if this was developed more, this could be used for NLP analytics and generative computer text.

Conclusion:

As a whole, WordSource can be used to quickly grasp the content matter of a body of text, whether it be a research paper, song lyrics, or messages from a business partner. The human language is so complex that representing patterns in it can help us gain insight into natural communication. This project was enjoyable to work on, and I will continue to use it for lyric generation.

References:

- NLTK Concordance Workaround
  https://simply-python.com/2014/03/14/saving-output-of-nltk-text-concordance/
- NLTK - Natural Language Toolkit
  https://www.nltk.org/
- D3.js - Javascript Visualization Library
  https://d3js.org/
- Chardet - Text File encoding and decoding Python library
  https://pypi.org/project/chardet/
- Web2Py - Full stack Python web application framework
  http://www.web2py.com/
- Vue.js - Frontend Javascript Framework
  https://vuejs.org/